The Journal of Open Source Software

t8code - modular adaptive mesh refinement in the exascale era

Johannes Holke[1*¶], Johannes Markert[1*], David Knapp[1*], Lukas Dreyer[1], Sandro Elsweijer[1], Niklas Böing[1], Chiara Hergl[1], Prasanna Ponnusamy[1], and Achim Basermann[1]

1 German Aerospace Center (DLR), Institute for Software Technology, Cologne, Germany ¶ Corresponding author * These authors contributed equally.

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

In partnership with

AMERICAN ASTRONOMICAL SOCIETY

This article and software are linked with research article DOI 10.3847/xxxxx <- update this with the DOI from AAS once you know it., published in the Astrophysical Journal <- The name of the AAS journal..

Summary

In this note we present our software library t8code for scalable dynamic adaptive mesh refinement (AMR) officially released in 2022 (Holke et al., 2022). t8code is written in C/C++, open source, and readily available at www.dlr-amr.github.io/t8code. The library provides fast and memory efficient parallel algorithms for dynamic AMR to handle tasks such as mesh adaptation, load-balancing, ghost computation, feature search and more. t8code can manage meshes with over one trillion mesh elements (Holke et al., 2021) and scales up to one million parallel processes (Holke, 2018). It is intended to be used as mesh management back end in scientific and engineering simulation codes paving the way towards high-performance applications of the upcoming exascale era.

Introduction

AMR has been established as a successful approach for scientific and engineering simulations over the past decades (Babuvška & Rheinboldt, 1978; Bangerth et al., 2007; Dörfler, 1996; Teunissen & Keppens, 2019). By modifying the mesh resolution locally according to problem specific indicators, the computational power is efficiently concentrated where needed and the overall memory usage is reduced by orders of magnitude. However, managing adaptive meshes and associated data is a very challenging task, especially for parallel codes. Implementing fast and scalable AMR routines generally leads to a large development overhead motivating the need for external mesh management libraries like t8code.

t8code is written in C/C++, open source, and the latest release can be obtained at https://dlr-amr.github.io/t8code (Holke et al., 2022). It uses efficient space-filling curves (SFC) to manage the data in structured refinement trees. While in the past being successfully applied to quadrilateral and hexahedral meshes (Burstedde et al., 2011; Weinzierl, 2019), t8code extends these SFC techniques in a modular fashion, such that arbitrary element shapes are supported. We achieve this modularity through a novel decoupling approach that separates high-level (mesh global) algorithms from low-level (element local) implementations. All high-level algorithms can then be applied to different implementations of element shapes and refinement patterns. A mix of different element shapes in the same mesh is also supported.

Currently, t8code provides implementations of Morton type SFCs with $1 : 2^d$ refinement for vertices ($d = 0$), lines ($d = 1$), quadrilaterals, triangles ($d = 2$), hexahedra, tetrahedra, prisms, and pyramids ($d = 3$). The latter having a $1 : 10$ refinement rule with tetrahedra emerging as child elements (Knapp, 2020). Additionally, implementation of other refinement patterns and SFCs is possible according to the specific requirements of the application.

The Journal of Open Source Software

The purpose of this note is to provide a brief overview and a first point of entrance for software developers working on codes storing data on (distributed) meshes.

For further information beyond this short note and also for code examples, we refer to our Documentation and Wiki (Holke et al., 2022) and our other technical papers on t8code (Becker, 2021; Burstedde & Holke, 2016, 2017; Dreyer, 2021; Elsweijer, 2021; Holke, 2018; Holke et al., 2021; Knapp, 2020; Lilikakis, 2022).

Fundamental Concepts

t8code is based on the concept of tree-based adaptive mesh refinement. Starting point is an unstructured input mesh, which we call coarse mesh that describes the geometry of the computational domain. The coarse mesh elements are refined recursively in a structured pattern, resulting in refinement trees of which we store only minimal information of the finest elements (the leafs of the tree). We call this resulting fine mesh the forest.

By enumerating the children in the refinement pattern we obtain a space-filling curve logic. Via these SFCs, all elements in a refinement tree are assigned an index and are stored in linear order of these indices. Information such as coordinates or element neighbors do not need to be stored explicitly, but can be recovered from the index and the appropriate information of the coarse elements. The less elements the input mesh has, the more memory and runtime are saved through the SFC logic. t8code supports distributed coarse meshes of arbitrary size and complexity, which we tested for up to 370 million input elements (Burstedde & Holke, 2017).

The forest mesh is distributed, that is, at any time, each parallel process only stores a unique portion of the forest mesh, the boundaries of which are calculated from the SFC indices; see Figure 1.

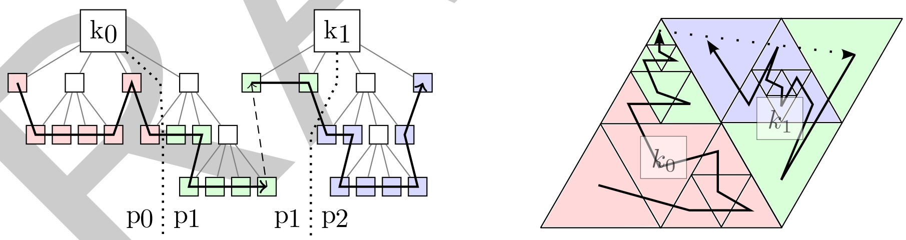

Figure 1: Left: Quad-tree of an exemplary forest mesh consisting of two trees (k_0, k_1) distributed over three parallel processes P0 to P2. The SFC is represented by a black curve tracing only the finest elements (leaf nodes) of each tree. Right: Sketch of the associated triangular mesh refined up to level three.

Interfacing with t8code

In this section we discuss the main interface of t8code and how an application would use it. While t8code offers various ways to interact with meshes and data, we restrict ourselves to the most important functionality here.

Every application is different and comes with their own requirements, data, and adaptation criteria. In order to support a wide variety of use cases, our core philosophy for t8code is to impose as few assumptions and to offer as much freedom as possible. We cater for this by applying the Hollywood principle: "Don't call us, we'll call you!". Whenever an application needs to interact with the mesh, e.g., adapting the mesh, interpolating data, etc., we offer suitable callback handlers.

The application developer implements custom callback functions and registers them via the t8code application programming interface (API). Any mesh specific details on how to access individual elements in the forest is opaque to the application and internally handled by t8code in an efficient manner. Of course, any typical application using hierarchical meshes needs to store data on the elements of a forest. This data might correspond to some simulated state variables, e.g., fluid velocity and temperature in a CFD simulation. In accordance to our core philosophy, the data is only loosely coupled with t8code's data structures. In order to properly access the application data in the callbacks, the data simply needs to be provided as a consecutive array with one entry per element enumerated in SFC order. For parallel applications, access to neighboring elements across parallel zones (ghost layer) is provided in a similar fashion.

Modularity & Extensibility

A distinct feature of t8code compared to similar AMR libraries is its high modularity achieved by decoupling high-level from low-level algorithms and coming along with it the support for arbitrary element shapes and refinement patterns. It also allows to combine different element shapes within the same mesh (hybrid meshes).

All high-level operations use the low-level algorithms only as a black box. For example, mesh adaption routines iterate through the mesh and when necessary call low-level algorithms for retrieving the children or the parent to refine or coarsen an element. In order to implement the logic of the adaption, however, no knowledge of the implementation details of these low-level functions is required.

Thus, for each individual tree we can simply replace the underlying implementation of the low-level algorithms (e.g. from tetrahedra to hexahedra) without affecting the high-level functionality. We achieve this by encapsulating all shape-specific element operations such as parent/child computation, face-neighbor computation, SFC index computation and more in an abstract C++ base class. The different element shapes and refinement patterns are then specializations of this base class. Hence, t8code can be easily extended - also by application developers - to support other refinement patterns and SFCs.

Moreover, this very high degree of modularity allows us to support an even wider range of non-standard additions. For example, the insertion of sub-elements to resolve hanging nodes (Becker, 2021) in quadrilateral meshes. Each quad element that has a hanging node is subdivided into a set of several triangles eliminating the hanging node.

Furthermore, we added support for holes in the mesh by selectively deleting elements (Lilikakis, 2022). This feature can be used to incorporate additional geometry information into the mesh. Similar to marking elements as getting refined or coarsened, we can additionally mark elements as getting removed. These elements will be eliminated completely from the SFC reducing the overall memory footprint.

Addionally, we support curved hexahedra with geometry-informed AMR (Elsweijer, 2021). Thus, information such as element volumes, face areas, or positions of interpolation/quadrature points in high order meshes can be calculated exactly with respect to the actual geometry. Another use case is to start with a very coarse input mesh and geometrically refine the mesh maxing out the performance benefits of tree-based AMR.

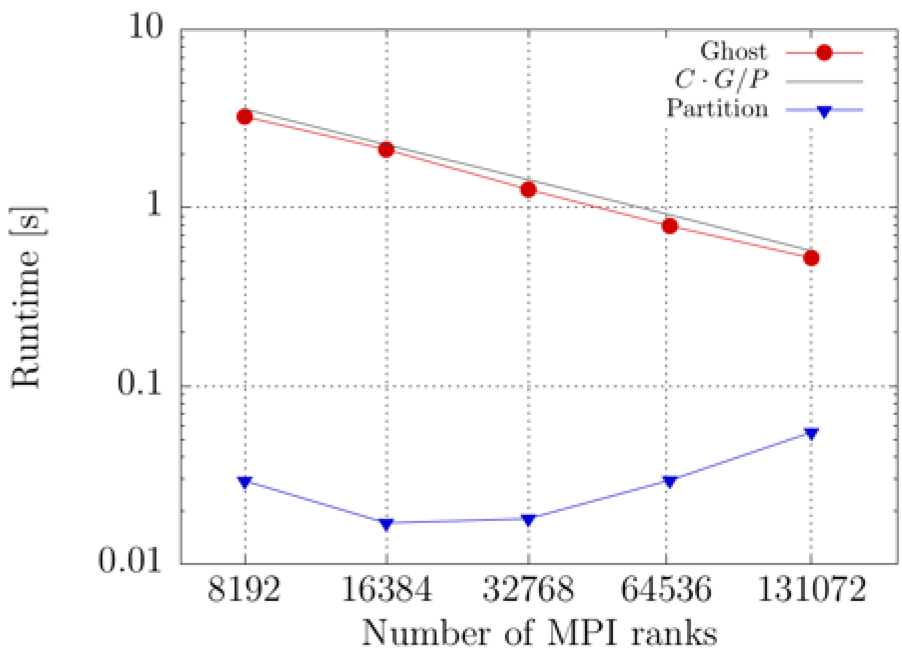

Figure 2: Strong scaling on JUWELS with tetrahedral elements. We plot the runtimes of Ghost and Partition routines with a refinement band from levels 8 to 10 after four time steps. Hence, the forest mesh consists of approximately 1.91 billion tetrahedra. As observed in the plot, we achieve perfect scaling for the Ghost algorithm in the number G/P of ghosts per process. The runtime of Partition is below 0.1 seconds even for the largest run. More details can be found in (Holke et al., 2021).

References

Babuvška, I., & Rheinboldt, W. C. (1978). Error estimates for adaptive finite element computations. *SIAM Journal on Numerical Analysis*, *15*(4), 736–754. https://doi.org/10.1137/0715049

Bangerth, W., Hartmann, R., & Kanschat, G. (2007). Deal.II—a general-purpose object-oriented finite element library. *ACM Trans. Math. Softw.*, *33*(4), 24–es. https://doi.org/10.1145/1268776.1268779

Becker, F. (2021). *Removing hanging faces from tree-based adaptive meshes for numerical simulations* [Master's thesis]. Universität zu Köln.

Burstedde, C., & Holke, J. (2016). A tetrahedral space-filling curve for nonconforming adaptive meshes. *SIAM Journal on Scientific Computing*, *38*, C471–C503. https://doi.org/10.1137/15M1040049

Burstedde, C., & Holke, J. (2017). Coarse Mesh Partitioning for Tree-Based AMR. *SIAM Journal on Scientific Computing*, *Vol. 39*, C364–C392. https://doi.org/10.1137/16M1103518

Burstedde, C., Wilcox, L. C., & Ghattas, O. (2011). p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. *SIAM Journal on Scientific Computing*, *33*(3), 1103–1133. https://doi.org/10.1137/100791634

Dörfler, W. (1996). A convergent adaptive algorithm for poisson's equation. *SIAM Journal on Numerical Analysis*, *33*(3), 1106–1124. https://doi.org/10.1137/0733054

Dreyer, L. (2021). *The local discontinuous galerkin method for the advection-diffusion equation on adaptive meshes* [Master's thesis, Rheinische Friedrich-Wilhems-Universität

Bonn]. https://elib.dlr.de/143969/

Elsweijer, S. (2021). *Curved Domain Adaptive Mesh Refinement with Hexahedra*. Hochschule Bonn-Rhein-Sieg. https://elib.dlr.de/143537/

Holke, J. (2018). *Scalable algorithms for parallel tree-based adaptive mesh refinement with general element types* [{PhD} thesis]. Rheinische Friedrich-Wilhelms-Universität Bonn.

Holke, J., Burstedde, C., Knapp, D., Dreyer, L., Elsweijer, S., Uenlue, V., Markert, J., Lilikakis, I., & Boeing, N. (2022). *t8code* (Version 1.0.0). https://doi.org/10.5281/zenodo.7034838

Holke, J., Knapp, D., & Burstedde, C. (2021). An Optimized, Parallel Computation of the Ghost Layer for Adaptive Hybrid Forest Meshes. *SIAM Journal on Scientific Computing*, C359–C385. https://doi.org/10.1137/20M1383033

Knapp, D. (2020). *A space-filling curve for pyramidal adaptive mesh refinement* [{M}aster's Thesis]. Rheinische Friedrich-Wilhelms-Universität Bonn.

Lilikakis, I. (2022). *Algorithms for tree-based adaptive meshes with incomplete trees* [Master's thesis, Universität zu Köln]. https://elib.dlr.de/191968/

Teunissen, J., & Keppens, R. (2019). A geometric multigrid library for quadtree/octree AMR grids coupled to MPI-AMRVAC. *Computer Physics Communications*, *245*, 106866. https://doi.org/https://doi.org/10.1016/j.cpc.2019.106866

Weinzierl, T. (2019). The Peano Software-Parallel, Automaton-based, Dynamically Adaptive Grid Traversals. *ACM Transactions on Mathematical Software*, *45*(2), 1–41. https://doi.org/10.1145/3319797